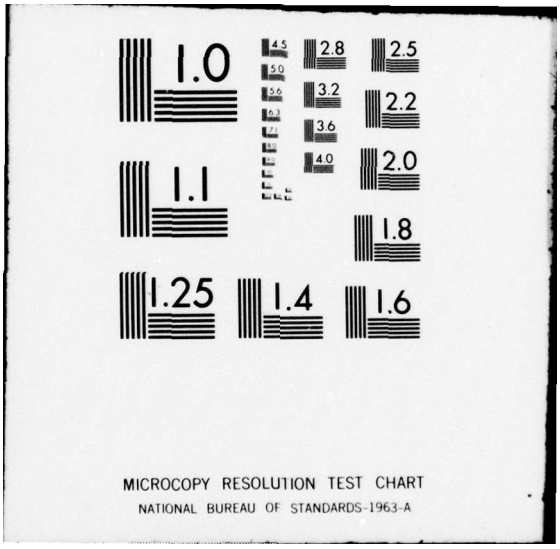
AD-A064 187 COMMAND AND CONTROL TECHNICAL CENTER WASHINGTON D C F/G 9/2
GUIDELINES FOR STRUCTURED CODING.(U)
SEP 78 J R SCOTT
UNCLASSIFIED CCTC-TM-185-78 SBIE-E100157 NL

| OF |
AD
A064187

END
DATE
FILMED
4 -79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

TECHNICAL MEMORANDUM
TM 185-78
1 SEPTEMBER 1978

LEVEL III

COMMAND & CONTROL TECHNICAL CENTER

GUIDELINES FOR STRUCTURED CODING

DEFENSE COMMUNICATIONS AGENCY

COMMAND AND CONTROL TECHNICAL CENTER

(9) Technical Memorandum Number (14) CCTC TM-185-78

(11) 1 Sep[redacted]78 (12) 50 p.

(6) GUIDELINES FOR STRUCTURED CODING.

(18) SBIE (19) AD-E100 157

PREPARED BY:

(10) JOHN R. SCOTT
Captain, USA
Project Officer

REVIEWED BY:

CHARLES W. DURIEUX
Technical Support
Office

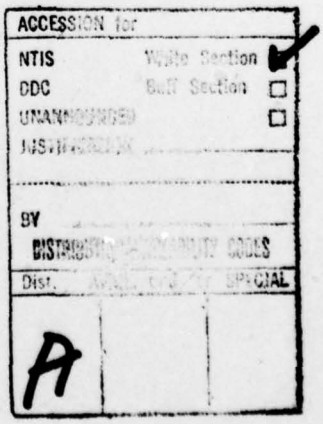
APPROVED BY:

B. E. MORRISS
DIRECTOR, CCTC

409 658

ACKNOWLEDGMENT

These guidelines are an abridgement of the guidelines produced for the US Air Force Rome Air Development Center by the IBM Corporation under contract F30602-74-C-0186.



CONTENTS

Section Page

ILLUSTRATIONS

(correct for p 1-1)

ABSTRACT

This manual provides guidelines for DCA/CCTC personnel writing computer programs using structured code. The guidelines are for handcoding of structured constructs in FORTRAN and COBOL, with additional instructions on module sizing and formatting.

SECTION 1. INTRODUCTION

The purpose of this manual is to provide guidelines for writing structured code within CCTC. These guidelines will provide CCTC project officers guidance for implementing structured coding using existing tools and techniques. It is felt that publication of guidelines at this time will serve to encourage experiments with this technology and will serve to standardize CCTC's approach. However, standardization is a secondary objective, since it is felt experience with these guidelines will allow CCTC to develop practical standards. Users of these guidelines are encouraged to provide comments to the Technical Support Office (C1l0), CCTC, Room BE685, The Pentagon, Washington, D.C. 20301.

SECTION 2. FUNDAMENTALS (BACKGROUND)

Structured coding is one of a large number of techniques which have evolved along with the concept of software engineering. Structured coding is not structured programming since structured programming is a label assigned to a series of software development techniques. Structured coding is the part of structured programming which has found almost universal acceptance by users of newer software development methodologies.

2.1 Structure Theorem

Structured coding is based upon the mathematically proven structure theorem which states that a proper program is one that meets the following requirements:

a. It has exactly one entry point and exactly one exit point for program control.

b. There are paths from the entry to the exit that lead through every part of the program; this means that there are no infinite loops and no unreachable code. This requirement is, of course, no restriction, but simply a statement that the structure theorem applies only to meaningful programs.

2.2 Basic Control Structures

The three basic control logic structures are defined as follows:

a. Sequence is simply a formalization of the idea that unless otherwise stated, program statements are executed in the order in which they appear in the program. This is true of all commonly used programming languages; it is not always realized that sequence is in fact a control logic structure. In flowchart terms, sequence is represented by one function after the other, as shown in figure 2-1.

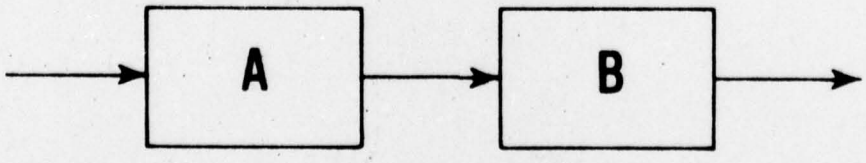


Figure 2-1. Flowchart for the Control Logic Structure Sequence

A and B are anything from single statements up to complete modules; the concern is only with the abstract idea of a proper program, regardless of its size and internal complexity. A and B must both be proper programs in the sense just defined (one entry and one exit). The combination of A followed by B is also a proper program, since it too has one entry and one exit.

Selection is the choice between two actions based on a predicate; this is called the IFTHENELSE structure. The usual flowchart notation for selection is shown in figure 2-2, where p is the predicate and A and B are the two functions.

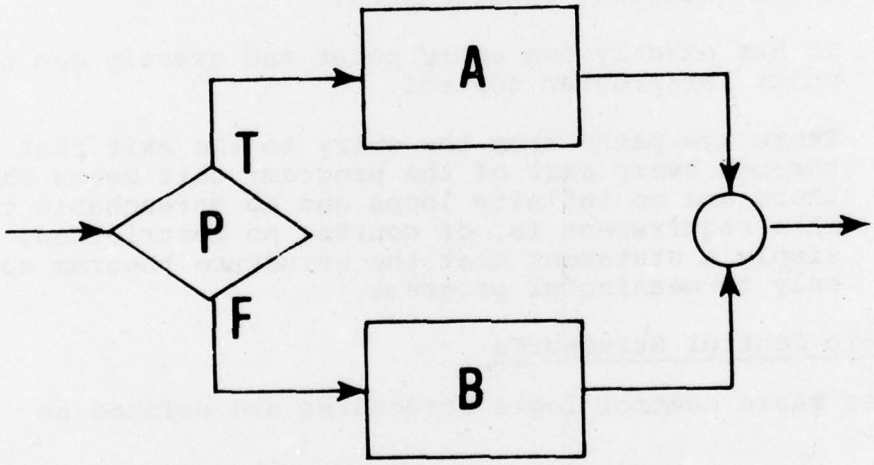


Figure 2-2. Flowchart for the Control Logic Structure Selection

The iteration structure, used for repeated execution of code while a condition is true (also called loop control), is the DOWHILE. In the flowchart in figure 2-3, p is the predicate and A is the controlled code.

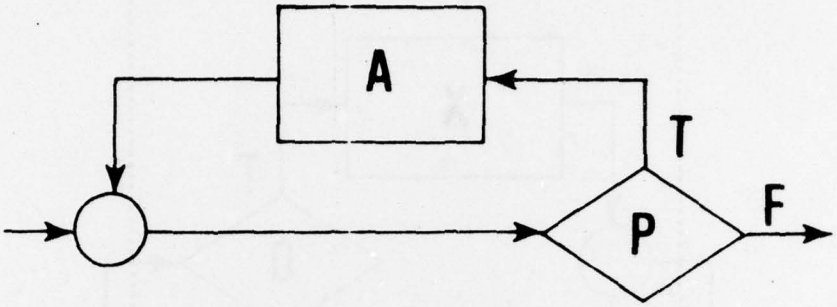


Figure 2-3. Flowchart for the Control Logic Structure Iteration, the DOWHILE

A fundamental idea is that anywhere a function box appears, any of the three basic structures may be substituted and still have a proper program. For example, the function box in figure 2-3 could be replaced with selection, producing the flowchart of figure 2-4. The dotted lines show where another structure has been substituted for a function. Flowcharts of arbitrary complexity can be built up in this way.

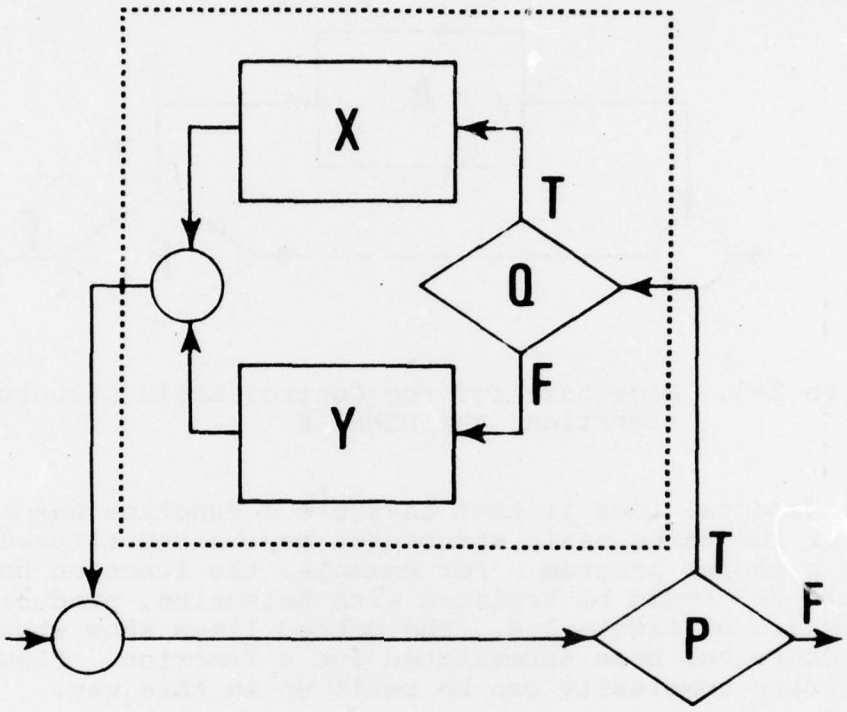


Figure 2-4. An Example of the Combination of Two Control Logic Structures, in which the Function Controlled by a DOWHILE is an IFTHENELSE

The ability to substitute control logic structures for functions and still have a proper program is basic to structured coding. This may also be called the nesting of structures.

2.3 Additional Control Structures

Although all programs can be written using only the three basic structures, it is sometimes helpful to utilize a few others.

The basic iteration structure is the DOWHILE, but there is a closely related structure, DOUNTIL, that is sometimes used, depending on the procedure that is to be expressed and on availability of appropriate language features. The flowchart is shown in figure 2-5.

The difference between the DOWHILE and DOUNTIL structures is that with the DOWHILE the predicate is tested before executing the function; if the predicate is false, the function is not executed at all. With the DOUNTIL, the predicate is tested after executing the function; the function will always be executed at least once, regardless of whether the predicate is true or false.

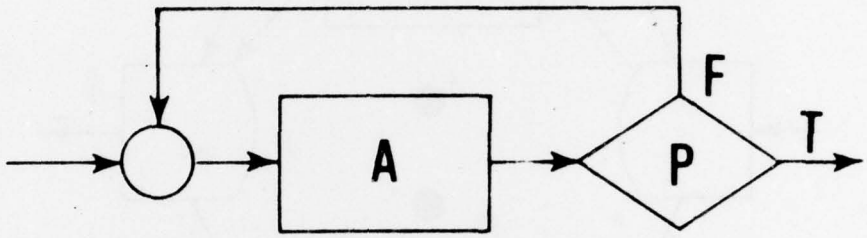


Figure 2-5. Flowchart for the Control Logic Structure Iteration, the DOUNTIL

It is sometimes helpful - from both readability and efficiency standpoints - to have some way to express a multiway branch, commonly referred to as the CASE structure. For example, if it is necessary to execute appropriate routines based on a 2-digit decimal code, it certainly is possible to write 100 IF statements, or a compound statement with 99 ELSE IFs but common sense suggests that there is no reason to adhere so rigidly to the three basic structures.

The CASE structure uses the value of a variable to determine which of several routines is to be executed. The flowchart is shown in figure 2-6. Observe that DOUNTIL and CASE are both proper programs.

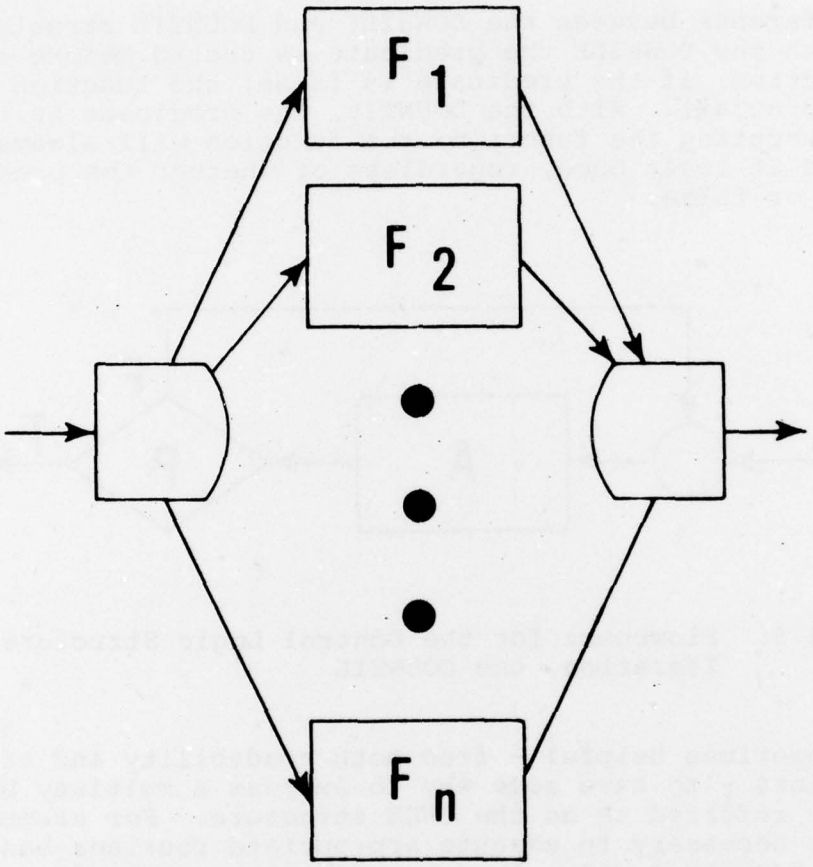


Figure 2-6. Flowchart for the CASE Control Logic Structure

SECTION 3. GENERAL GUIDELINES

3.1 Introduction

Guidelines for structured coding which are language independent are presented in this section. These guidelines are recommendations which would be considered by project managers. However, deviation will occasionally be required to satisfy unique project requirements.

Four approaches are possible when developing structured coding for existing nonstructured languages. The techniques are:

a. Use those structuring capabilities which are present in the language.

b. Simulate structuring capabilities using the features of the language including the unconditional branch and the commenting features, as necessary.

c. Use a precompiler to translate top down structured code into the target high level language source code.

d. Modify the language syntax to include the necessary structuring features and make these modifications a part of the language compiler.

In this manual a combination of techniques a. and b. above is used to define standards and guidelines for each language. Precompilers are available in CCTC for COBOL and FORTRAN, but their use should be considered only after examination of their impact on future developments. The last possibility, that of modifying the languages, is a long range objective and is not addressed.

3.2 Precompilers

A structured precompiler is a computer program which accepts programs containing structures not native to a language and produces output which is acceptable to the language's compiler. By using a precompiler, it is possible to implement structured coding by directly coding the structured constructs and eliminating the error prone process of manually translating the structured constructs. In addition to time savings, a precompiler also provides compliance checking. A precompiler's

principal disadvantage is the lack of standards for the input language. This means that the user of a precompiler must depend upon the continuing availability of the particular precompiler for the life of project software. In other words, the precompiler becomes part of the application system. Precompilers are also inconvenient to use while debugging since there diagnostics refer to the precompiler output not the original structured source code. This requires that programmers work with the intermediate code, which may not be as readable.

A number of precompilers for structured code have been developed and versions for GCOS COBOL and FORTRAN are available in CCTC for evaluation.

3.3 Language Independent Guidelines

The following language independent guidelines must be followed in order to implement structured coding:

a. Every code segment should contain a single entry and a single exit.

b. Explicit branching (GOTO type instructions) is discouraged, but there are occasions when judicious use will result in substantial improvement. These deviations should be accommodated and appropriately documented. Also, hand coding of the structured constructs will result in explicit branches in the implementation language.

c. In free format languages, only one statement per line of code is permitted.

d. Indention to indicate the span of control of a structure must be used.

These guidelines are language independent. They are based on the structure theorem, or to make programs easier to read.

3.4 Language Dependent Guidelines

Language dependent guidelines (section 4) provide the methods for simulating control logic structures. In addition, section 4 also contains other recommendations which may be considered. as guidelines. These cover such items as indentation rules, grouping of data, data formats, etc. The most important

consideration with respect to guidelines is not that the ones described be implemented exactly as indicated but rather that, for a given project, conventions be established for the indicated areas, and then applied uniformly throughout the entire project.

SECTION 4. LANGUAGE DEPENDENT GUIDELINES

4.1 Introduction

This section provides guidelines for structured coding software development in HIS FORTRAN, and ANSI COBOL. Preceding each lanaguage is a general discussion of the structuring capabilities of the language. This is followed by a subsection which shows how the basic control logic figures may be implemented and how the nested inclusion of code segments may be achieved using only those features which are native to the language itself. The final section of each language deals with other coding guidelines whose primary purpose is to enhance readability and maintainability of the programs that are produced.

In the coding examples which are shown in the language sections, a lower case letter enclosed in parentheses(e.g., (p), (q)), represents a conditional expression which is written in the syntax of the language under discussion. The word "code" written in lower case followed by a capital letter (e.g., code A, code B) represents a block of code which may consist of any valid language statements which are compatible with structured coding concepts.

The individual language subsections are written in such a way as to be independent of each other in the assumption that a user would reference only the language that was of interest to him. Thus, it is necessary to repeat certain material in each subsection.

It has also been assumed that the reader of any language subsection is familiar with both the syntax and the terminology in the language manual which describes the language. In each language section an attempt has been made to use the language manual terminology when discussing the various language features.

4.2 ANS FORTRAN

4.2.1 General Comments. This manual is based on the current WWMCCS version of Honeywell GCOS FORTRAN, which is based upon the ANSI X3.9-1966 standard. The new X3.9-1978 standard incorporates features, such as an IF-THE-ELSE, which improve its utility for structured coding. However, there are no plans to provide WWMCCS a 3.9-1978 compiler. Although FORTRAN does not support the basic structured coding figures, it accommodates simulation of these figures. However, problems will be encountered in simulating the structured figures.

First, there is no automated verification of the correctness of the simulated figures, and second, the branching which must be used in these figures adds complexity during the program coding and checkout phases.

The logical IF (as distinguished from the arithmetic IF) statement in FORTRAN does not provide the IFTHENELSE capability, i.e., two possible paths, but rather allows conditional execution of only one statement provided the logical expression tested is true. The IFTHENELSE figure thus must be simulated with a logical IF and one or more GOTO statements.

The DO statement in FORTRAN is an indexing type of looping statement which is of the DOUNTIL form since the test occurs at the end of the loop. The capability of iteration based on whether a condition is true or false does not exist. Therefore the DOUNTIL and DOWHILE control figures must be simulated with a logical IF and GOTO statements.

FORTRAN has no implementation of the CASE figure as such. However, use of the computed GOTO along with the unconditional GOTO statements provides a functional basis for the simulation of the CASE figure.

The following subsections detail a simulation of the basic structured coding figures (subsection 4.2.2) and provide a set of suggested language coding conventions (subsection 4.2.3) which are intended to have a minimal impact on the FORTRAN user. They are all intended to achieve basic goals: to produce programs which are easy to write and debug, easy to read and understand, and easy to maintain and modify.

In summary the deficiencies in FORTRAN which affect its top down structured coding capability are:

a. The lack of an INCLUDE capability

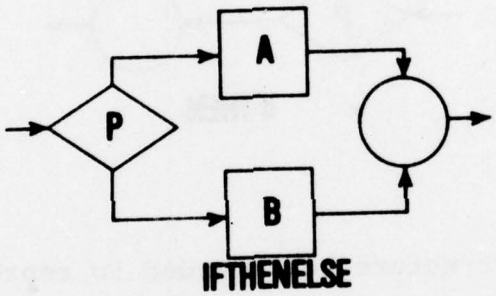
b. The lack of both the DOWHILE and DOUNTIL control structures

c. The lack of the IFTHENELSE control structure.

4.2.2 Structured Coding in ANS FORTRAN. As mentioned previously, a basic problem in simulating the structured coding figures using the features of FORTRAN lies in the fact that there is no automated verification of the structured figure's integrity. Thus, an incorrectly coded structured figure might appear correct in form. No solution to this problem is proposed for FORTRAN figure simulation.

In defining each standard figure in this section, the goal has not been to attempt to minimize the number of lines written; rather it has been to strive for clarity so that the identity of each figure can be understood without any doubt.

4.2.2.1 IFTHENELSE Figure. The IFTHENELSE figure causes control to be transferred to one of two functional blocks of code (A or B) based on the evaluation of a logical expression (p). Since the logical IF statement in FORTRAN allows conditional execution of a single statement, provided the logical expression tested is true, the IFTHENELSE figure is simulated in FORTRAN using both a logical IF statement and GOTO statements. The flowchart for the IFTHENELSE figure is:



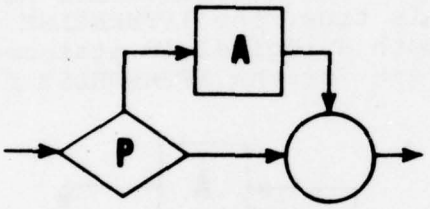
IFTHENELSE

The IFTHENELSE figure should be constructed so that the "true" conditional code (a) immediately follows the IF statement. In order to implement this in FORTRAN, it is more desirable to test for the negative of the condition desired by using a ".NOT." (a positive test would necessitate the use of an additional GOTO statement). The code structure recommended to represent the IFTHENELSE figure is:

```
      IF (.NOT.(p)) GO TO 0010
     code A
     GO TO 0020
C    ELSE
0010 code B
0020 CONTINUE
C    ENDIF
```

Statements within the two clauses including the GOTO terminating the first clause and the CONTINUE statement terminator for the figure should be indented two columns. The ELSE and ENDIF comment lines which aid in locating the end of the functional blocks of code should be aligned with the IF.

The ELSE in the IFTHENELSE figure is optional and if not used, the flowchart for this figure would be reduced to:



IFTHEN

and the code structure recommended to represent this is:

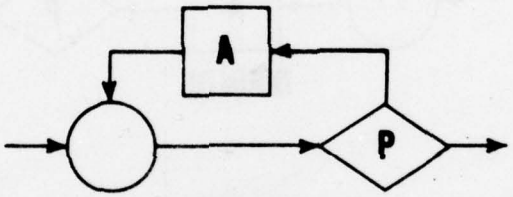
```
      IF (.NOT.(P)) GO TO 0010
        code A
0010  CONTINUE
C     ENDIF
```

If code A consists of a single statement the logical IF statement, which is a part of the FORTRAN language, may be used. The recommended format is as follows:

```
      IF (P)
     1   code A
C     ENDIF
```

4.2.2.2 DO Figures. The DO figures allow iterative execution of functional block of code (a) based on a logical expression (p). If the test is made prior to the execution of code A it is a DOWHILE figure. If it is made after code A it is a DOUNTIL figure. The FORTRAN DO is essentially a specialized DOUNTIL and its use to simulate the DO figures is very clumsy. With the FORTRAN DO, execution continues as long as an index is not incremented past a test value; however, the DO statement is a command to execute, at least once, the statements within its range. The DO figures can easily and more understandably be simulated in FORTRAN using a logical IF statement and GOTO statements.

4.2.2.2.1 DOWHILE. It is recommended that the DOWHILE figure be simulated in FORTRAN using a logical IF statement and GOTO statements. The flowchart for the DOWHILE figure is:



DOWHILE

Note that the logical expression (p) is tested prior to each execution of the functional block of code (a) including the first.

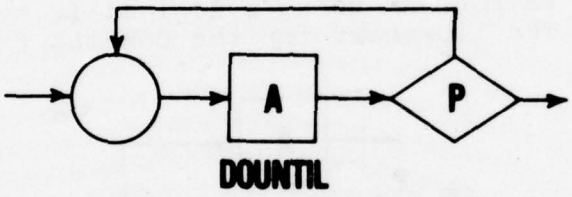
Although this figure can be coded using either a positive conditional test or a negative conditional test, the positive conditional test approach is recommended for FORTRAN implementation. The code structure recommended to represent the DOWHILE figure is:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
C           DO    WHILE  (P)
                  GO TO  0020
     0010 code A
     0020 IF (P) GO TO 0010
C           ENDDO
```

The logical expression (p) on the DOWHILE comment line is recommended but could be deleted at the user's option.

Statements within the figure should be indented two columns from the DOWHILE and ENDDO comment lines which aid in locating the beginning and end of the figure.

4.2.2.2.2 DOUNTIL. It is recommended that the DOUNTIL figure also be simulated in FORTRAN using a logical IF statement and a GOTO statement. When looping under control of an index, however, a FORTRAN DO might be an appropriate choice (refer to the next subsection). The flowchart for the DOUNTIL figure is:



DOUNTIL

Note that the logical expression (p) is tested after each execution of the functional block of code, so that code (a) is always executed at least once.

The recommended simulation of this figure requires that the conditional test on the looping variable be negated as indicated in the example below. The negative condition is achieved by applying a ".NOT." to the desired logical expression. The code structure recommended to represent the DOUNTIL figure is:

```
C     DO UNTIL (p)
  ØØLØ   code A
         IF (.NOT.(p)) GO TO ØØ1Ø
C     ENDDO
```

The logical expression (p) on the DOUNTIL comment line is recommended but could be deleted at the user's option.

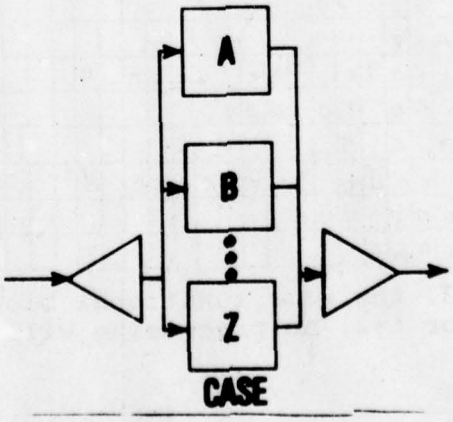
Statements within the figure should be indented two columns from the DOUNTIL and ENDDO comment lines which aid in locating the beginning and end of the figure.

4.2.2.2.3 The FORTRAN DO. The FORTRAN DO statement is a command to execute, at least once, the statements that physically follow it, up to and including the numbered statement indicating the end of the DO's range. Since the FORTRAN DO is essentially a specialized DOUNTIL figure, it is desirable that specific guidelines be stated for using this FORTRAN capability. The code structure recommended when using the FORTRAN DO statement is:

```
      DO  ØØ1Ø (indexing specification)
          code A
   ØØ1Ø CONTINUE
C     ENDDO
```

The statements within this specialized DOUNTIL figure should be indented two columns from the DO statement and the ENDDO comment line which aids in determining the end of the figure.

4.2.2.3. CASE Figure. The CASE figure causes control to be passed to one of a set of functional blocks of code (A, B, ..., Z) based on the value of an integer variable i, equal to (1, 2, ..., m). It is recommended that the CASE figure be simulated in FORTRAN using a computed GOTO statement, GOTO statements and a single collector (CONTINUE statement) at the end of the figure. The flowchart for the CASE figure is:



Some FORTRAN compilers provide that if the value of I is outside the range $1 \leq i \leq m$, the next statement is executed. However, with ANS FORTRAN the computed GOTO statement is undefined if it is not within the range. Therefore in ANS FORTRAN, i should be tested prior to entering the CASE control logic structure. Out of range values should result in control being transferred to the default code.

The default code and the "GOTO ØØnØ" immediately following the computed GOTO statement are provided for use with compilers that provide for execution of the next sequential statement when i is not within the range of the computed GOTO.

Statements within the figure are indented two columns from the CASENTRY and ENDCASE comment lines which aid in determining the beginning and end of the figure. Statements within each case are indented two columns from each CASE comment line.

If the functional blocks of code are identical for more than one case, the appropriate entries in the computed GOTO statement should contain the same number. Further, if no action is to be performed for specific values of i. the appropriate entries should point to the end of the figure. Consider the following example.

```
C     CASENTRY
      GO TO (0010,0040,0010,0020,0030),I
0001  default code
      GO TO 0040
C     CASES 1 AND 3
0010  code for cases 1 and 3
      GO TO 0040
C     CASE 4
0020  code for case 4
      GO TO 0040
C     CASE 5
0030  code for case 5
0040  CONTINUE
C     ENDCASE
```

For i=1 and i=3, the same functional block of code will be executed and for i=2, no processing will be performed.

4.2.2.4 INCLUDE Capability. The capability of nesting blocks of code within other code blocks is a necessity for top down programming. This is most easily achieved if the language has a compiler directing instruction such as INCLUDE or COPY. In the case of ANS FORTRAN this type of statement does not exist and therefore the effect of nesting may be simulated by the use of nested CALLs of subroutines. However, since the linkages generated by CALL statements may be costly in terms of overhead, two other standard alternative simulations of the nested INCLUDE capability are presented. The first may be used if the included code segment appears in only one place in the program. It is written as follows:

```
C     INCLUDE function name
      GO TO 0010
0020  CONTINUE
C     END INCLUDE
```

The function name on the INCLUDE comment should be meaningful enough to indicate the processing performed by the out-of-line code. The out-of-line code then terminates with an explicit GOTO to the CONTINUE statement. If the same functional process is to be simulated as an INCLUDE in more than one place in the program, the assigned GOTO may be used to return control from the out-of-line code as follows:

```
C     INCLUDE function name
      ASSIGN 0010 TO I
      GO TO 1000
 0010 CONTINUE
C     END INCLUDE
      .
      .
      .
C     INCLUDE function name
      ASSIGN 0020 TO I
      GO TO 1000
 0020 CONTINUE
C     END INCLUDE
      .
      .
      .
```

The code which begins at statement 1000 when terminated with the following assigned GOTO statement:

```
      GO TO I, (0010,0020)
```

then returns control to the correct point to complete the simulation of nested INCLUDE.

The major problem encountered with the lack of an INCLUDE directive is related to the adverse effect this has on both the debugging procedures. Ideally the most current listing of a block of source code is filed in a notebook where is may be examined by any person who wishes to do so. This implies a mechanism for storing such blocks as individual entities of these small entities. A change can be made to any block without the necessity of passing all existing code through the editing routine and the filing of the revised listing in the library does not require replacement of other module lisings. However, the input to the FORTRAN compiler requires

that all of these individual subroutine blocks be gathered into a single sequential data set before being fed into the compiler. It is this capability which is supplied with the INCLUDE or COPY and the lack of it means that the program must be developed as a single sequential data set. In order to handle this problem various solutions outside the scope of the language have been implemented, such as precompilers, linkage editor INCLUDEs, or a data set concatenation capability within the operating system.

4.2.3 Additional Recommended Coding Conventions

4.2.3.1 Restricted FORTRAN Statement Usage. In order to maintain structured coding concepts, it is recommended that certain allowable FORTRAN statements generally not be used except as required in the previous definition of the standard program figures and summarized below. For the most part, an attempt is made to preclude unconditional branching not necessitated by standard program figure definition.

The GOTO statement is used in the definition of the following standard program figures: IFTHENELSE, DOWHILE, DOUNTIL and CASE. The computed GOTO statement is used in the definition of the CASE standard program figure. It should be an objective not to use these statements except in those figures.

The ASSIGN and ASSIGNED GOTO statements provide an unconditional branching capability. The arithmetic IF statement is not necessary because the IFTHENELSE standard program figure, with nesting sometimes required, will provide the same capability. Use of these FORTRAN statements should be avoided.

The recommended use of the DO statement as a specialized DOUNTIL, is covered in a previous subsection. Other usage of the DO is not recommended.

The CONTINUE statement is used in the definition of the IFTHENELSE and CASE standard program figures. In addition, it is sometimes required by a DO (specialized DOUNTIL) statement. No other use of the CONTINUE should be necessary.

4.2.3.2 Program Organization. These conventions provide for the organization of a FORTRAN source program into a set of segments for compilation. Any FORTRAN program requires

a certain ordering of the statements within the program. A suggested further restriction to that ordering for the sake of readability, clarity, and consistency appears below.

a. If this is a subprogram, the first card must be a FUNCTION, SUBROUTINE, or BLOCK DATA statement.

b. Any COMMON statements, each followed by all type, DOUBLE PRECISION, and EQUIVALENCE statements related to it follow. No dimension information is to appear on a COMMON statement. The COMMON statement will be used only to declare the order of arrays and variables within the COMMON. Blank COMMON is to be declared first, followed by all labeled COMMONs in alphabetical order.

   Any explicit specification (type) statements and DOUBLE PRECISION statements will be arranged in alphabetical order of the variables or arrays within each of the types. They will be defined in the following order: COMPLEX, DOUBLE PRECISION, REAL, INTEGER, and LOGICAL. All dimensioning information should be included on the type or DOUBLE PRECISION cards. All variables or arrays should be explicitly declared, and the DIMENSION statement should not be used in place of a type of statement.

Following each type or DOUBLE PRECISION statement, any EQUIVALENCE statements required for that type statement are included. A blank comment card should be used before and after the EQUIVALENCE statements to set them off from the surrounding definitions.

c. Once all COMMON declarations are made, the program local declarations are made using the same conventions.

d. Following all program local declarations, all EXTERNAL declarations will be made.

e. Any DATA statements for program local arrays and variables follow.

f. All FORMAT statements follow.

g. Any statement function definitions come next and complete the nonexecutable code.

h. Segments containing executable code follow in order. The last segment must contain an END card.

i. If desired, subprograms may follow as part of a multiple compilation. The organization of each subprogram should follow the rules given above.

4.2.3.3 Comments. Comments should be used to enhance the readability and understanding of a program (e.g., to define variables or their special settings). In general, when they are used they should be grouped together as a prologue to the code segment. If they must be interspersed within the code, they should be inserted as a block which begins in a column near the middle of the page (e.g., column 35 or 40) so as not to interfere with the indentation and readability of the program proper which may be scanned near the left margin. Blank comment cards should be used when they enhance readability.

4.2.3.4 Statement Numbering. As much as possible, statement numbers are to proceed from lowest to highest as a program is read. It is recommended that statement numbers be four digits long, be placed in columns 2-5, and be incremented by 10 rather than be consecutive.

4.2.3.5 Continuation Cards. ANS FORTRAN permits up to 19 successive continuation cards per statement. The continuation column should be used to indicate the order of the cards. This may be done by placing a numeric character in column 6 (the continuation indication column) in ascending sequence (i.e., 1-9) and if additional characters are necessary, using in order the alphabetic characters A-J.

The body of the continuation card should be coded so as to enhance the readability of the program unit. In the following subsections are some suggestions in regard to special cards, but, in general, no continuation card should contain information to the left of the statement identifier on the first card.

4.2.3.6 Statements. Each statement should begin in an even-numbered column. Nonexecutable statements should begin in column 8. If the program is a subprogram, the FUNCTION, SUBROUTINE, or BLOCK DATA statement should begin in column 8, and the corresponding END statement should also begin in column 8; in these cases, the first executable statement will begin in column 10. Otherwise, the first

executable statement of a program should begin in column 8, as should the END card. Succeeding executable statements are indented according to the coding examples in the standard program figures section and the rules given below.

4.2.3.6.1 Assignment Statements. If a statement is continued, the second and following lines should be indented by six columns. For example:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
      C O N V = ( Q N M * ( 2 . Ø * * P A B S F T ) * F G A I N ) /
      1 (S E I S N O * R T S T A * R P R I M E * R A T I O )
```

In a continued assignment statement, the first card (and each card which will be followed by another continuation card) should have an operator as the last character. As much as possible, all parenthetical expressions should be on a single card. In no case should a left parenthesis be the last character on a card which is to be continued.

4.2.3.6.2 COMMON Statements. The COMMON card will begin in column 8 with the identifier followed by a blank in column 14 and a "/" in column 15. The next six columns are reserved for the label which will be left justified in this field. Column 22 will contain another "/" and column 23 will contain a blank. If blank COMMON is desired, code the slashes but leave the label field blank. Commas will appear in columns 30, 37, and 44. If a continuation card is necessary, a comma will appear in column 23 on that card. The names will be left justified within each field.

4.2.3.6.3 Type Statements. Dimension information for each array is to be contained on a single type card. A card will contain one and only one name. A type statement may be continued, thus allowing 20 names to be declared for each statement.

The type statement will be coded beginning in column 8. The name should begin in column 16. If it is a continuation card, a comma should be inserted into column 15. When using a DOUBLE PRECISION statement, the name should begin in column 25. If a continuation card is required, a comma should be inserted into column 24.

4.2.3.6.4 FORMAT Statements. The FORMAT identifier will be coded beginning in column 8. The first parenthesis will

appear in column 16, and the format information itself begins in column 18. As much as possible, a position code and its associated format code should stand alone on a card. Continuation cards should be used liberally. For example:

```
      8Ø6Ø    FORMAT  (  1HØ,2ØX,8HMONTH  =  ,I2,
     1                          4X,6HDAY   =  ,I2,
     2                          4X,7HYEAR  =  ,I4)
```

4.2.3.6.5 READ or WRITE Statements. In all cases a READ or WRITE statement should have its file reference contained in a variable. The use of hard coded file references is discouraged from the standpoint of visibility and parameterized coding. The variable should be contained in a common block and initialized in a block data.

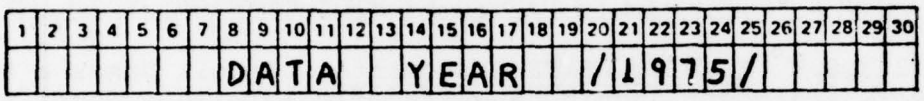
The list portion of the READ or WRITE statement must be expressed in as simple terms as possible. Liberal use of blanks and continuation cards is encouraged in order to increase readability.

4.2.3.6.6 IF Statements. Multiple conditions in the predicate of an IF statement should occur on separate cards, with an operator occurring as the final item in each card to be continued. For example:

```
      IF (A.GT.B  .OR.
     1   C.LT.D  .AND.
     2   X.EQ.J
     3  )  GO TO  ØlØØ
```

Notice also that the GOTO statement follows the closing parenthesis.

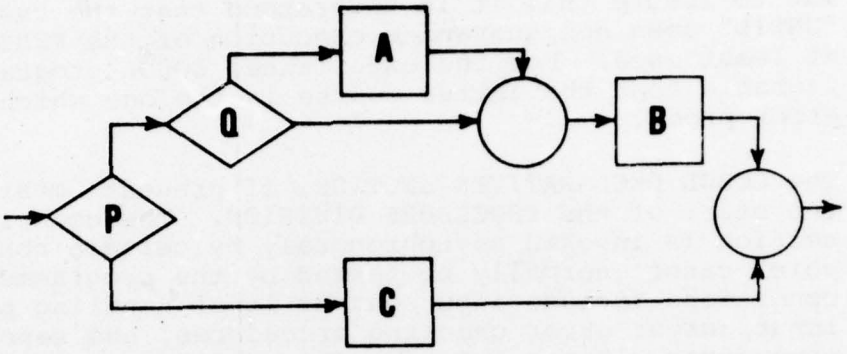
4.2.3.6.7 DATA Statements. Only one variable may be specified on a DATA statement card. The DATA identifier will be coded beginning in column 8. The variable name will be coded beginning in column 14. The slash indicating the beginning of the data will be coded in column 20. For example:



4.3 ANSI COBOL

4.3.1 General Comments. The two statements IF....ELSE and PERFORM, which are part of the COBOL supply the basic structuring capability of the language. The first represents the IFTHENELSE control figure while the PERFORM permits the looping required by the DOWHILE/DOUNTIL construct. In addition, the GO TO.... DEPENDING ON is readily adapted to the simulation of the CASE statement, the COPY assists in top down programming, and the language's SEARCH statement recognizes the potential utility of such a control logic structure. However, even with all of these features, the language does have certain deficiencies in the implementation of structured coding technology.

For instance, because of the way the period delimiter affects the syntax of the language, it is not possible to implement the following flowchart using only the COBOL IF statement without duplicating the sequence of statements in code B or duplicating the test on (p):



Three alternatives, shown in subsection 4.3.2.1, indicate how the processing may be achieved. However, with the use of a specific delimiter such as an END-IF, the problems encountered in implementing the above flowchart could be overcome.

The looping capability is achieved in COBOL with the PERFORM statement. All such loops are of the DOWHILE type since the looping condition is tested prior to execution of the code within the loop. This statement permits repetitive code execution under the following conditions. First, the programmer can indicate that the loop is to be executed a specified number of times. Second, an indexing type of loop can be requested with the PERFORM....VARYING option and finally the PERFORM....UNTIL option exists which is equivalent to the structured coding DOWHILE control figure. The selection of the word UNTIL to indicate loop termination conditions in COBOL is an unfortunate one since in structured coding terminology not only does it imply a different flowchart but the semantics of the looping control is the negative of the structured coding DOWHILE. Thus (ignoring temporarily the flowchart differences between WHILE and UNTIL), the statement "perform a loop WHILE something is true" translates semantically to "perform a loop UNTIL something is not true." Two options are open to the COBOL programmer. One is to simulate the DOWHILE by negating the conditional logic as follows:

```
PERFORM....UNTIL (NOT p).
```

The second is to continue to think of the looping condition in the UNTIL logic to which the COBOL programmer is accustomed but to insure that it is understood that the use of the word "UNTIL" does not guarantee execution of the PERFORMed code at least once. For the experienced COBOL programmer it is probable that the latter course is the one which is less error prone.

The COBOL DECLARATIVES SECTION, if present, must appear at the start of the PROCEDURE DIVISION. The code in this section is invoked asynchronously by certain conditions which cannot normally be tested by the programmer. These conditions include input/output label handling procedures, input/output error checking procedures, and report writing procedures. Since these blocks of code are out-of-line, they involve a transfer of control which is invisible to the programmer. Such interruptions of sequential control are usually undesirable in structured coding. However, because of the utility of the DECLARATIVES SECTION, no attempt has been made to restrict its usage, particularly since ANS COBOL requires that control automatically return to the statement following the one which caused the asynchronous interrupt.

The COPY and PERFORM statements, in spite of their limitations, are helpful in implementing top down programming. It is essential in top down programming to have the capability of nesting relatively small blocks of code within other such blocks. However, the COPY compiler directive cannot fulfill this function because it is limited to a single level (i.e., code which is copied cannot have a COPY statement within it). Therefore, it is necessary to simulate this requirement with PERFORM statements since these can be nested. One method of simulating the required nesting is not to permit a COPY statement in any segment which is invoked by the PERFORM statement. Then, after the top module is coded, COPY statements may be used to direct the compiler to include in its compilation the various PERFORMed paragraphs and thus overcome the limitations on the nesting of COPY statements. It should be noted that this type of organization implies that the modules which are copied, are located in a library system of some sort.

Finally COBOL as with many higher level languages, has a free format syntax. This permits the programmer to write statements in a continuous prose format instead of requiring the more desirable format of each new statement starting a new line of code and thus enhancing the readability of the structured code. Finally, in the examples which follow, the parentheses which enclose the conditional expressions are optional.

In summary, the deficiencies of COBOL which affect structured coding are:

a. The limitation of the IF....ELSE statement which restricts the nesting capability

b. The lack of a DOUNTIL capability

c. The free form of the language

d. The lack of specific delimiters such as END-IF, END-READ etc.

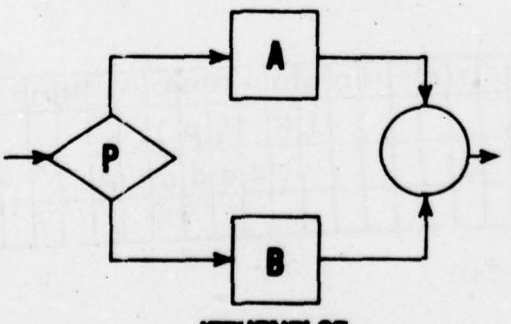
e. The inability to place the repetitive code of a looping operation in-line

f. The limitation of the COPY statement to a single level.

It must also be noted that the COBOL PERFORM...UNTIL has a meaning opposite to the DOUNTIL. Although not a language deficiency, this can be confusing to many programmers.

4.3.2 Top Down Structured Programming in ANS COBOL

4.3.2.1 IFTHENELSE Figure. The IFTHENELSE figures causes control to be transferred to one of two functional blocks of code (A or B) based on the evaluation of a logical expression (p). This is identical to the manner in which the COBOL conditional statement operates to this control logic structure does not have to be simulated. The flowchart for the IFTHENELSE figure is:

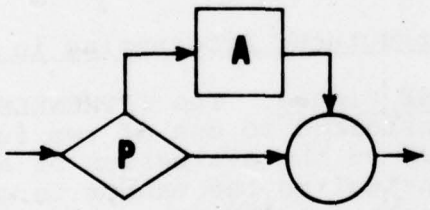


IFTHENELSE

The implementation of this control structure using the conditional statement is as follows:

```
IF (p)
    code A
ELSE
    code B.
```

The ELSE in the IFTHENELSE figure is optional and if not used, the flowchart reduces to:

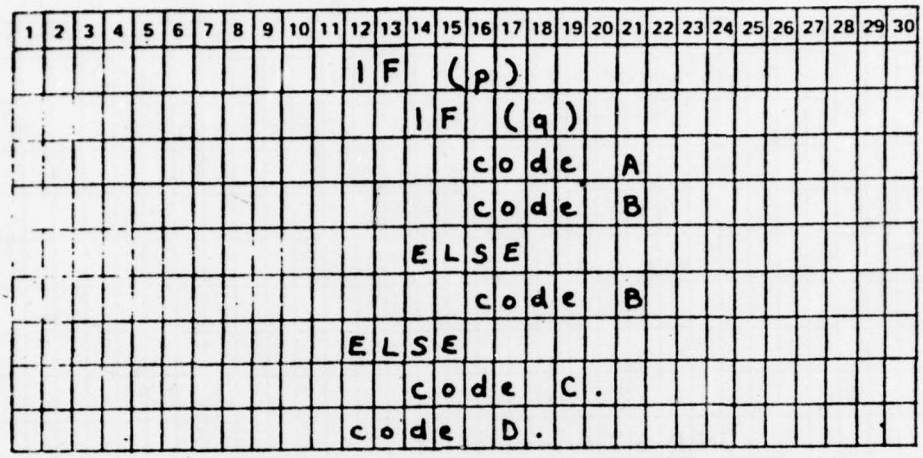


IFTHEN

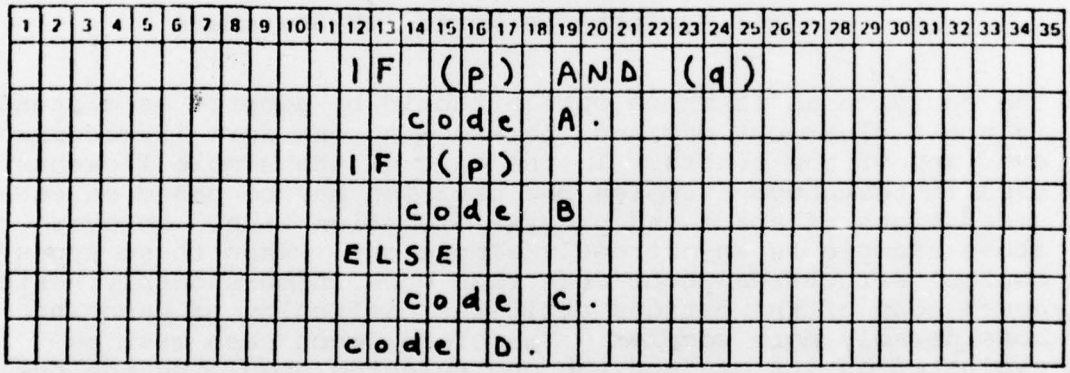
Since the ELSE is also optional in the COBOL conditional statement, the code becomes:

```
IF (p)
    code A.
```

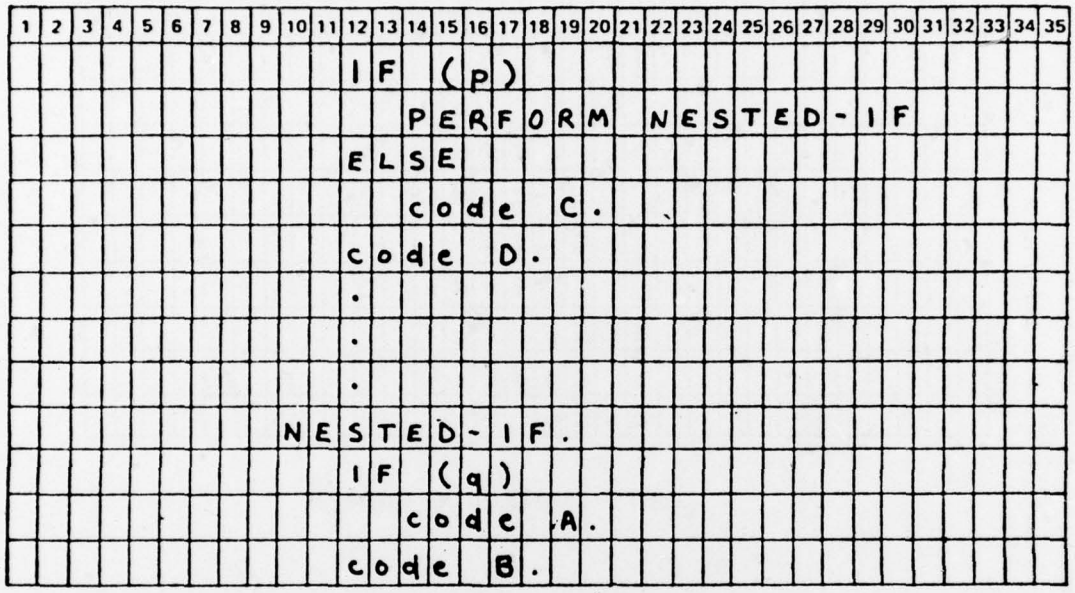
In order to overcome the nesting limitation resulting from the flowchart discussed in subsection 4.3.1 (page 4-17) the following options are available:



(a) Duplicate Code B



(b) Duplicate the Outermost Test on (p)

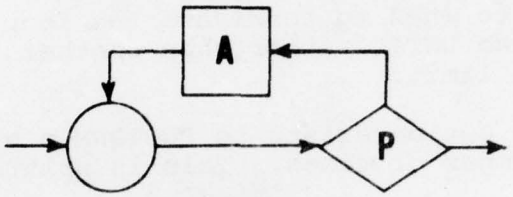


(c) Perform the Nested IF Statement

The decision as to which option should be adopted as a standard is not a clear cut one and therefore no one form is recommended over any of the others. In the case of the simple flowchart used in the above examples the decision may be based on considerations of space and program execution time. However, the above example is an extremely simple one. When these types of control structures occur more than once in more deeply nested code, none of the options avoid the difficulty of becoming considerably more complex. Therefore, each case must be evaluated on its own and the one selected should be the one which is the most readable and understandable. Indentation should be as indicated in the examples. As may be noted, code is indented beneath the IF and ELSE in order to emphasize the span of control of these verbs. Since no unique terminator is available, the end of the figure is indicated the the start of a new code block in the same column as the IF and ELSE.

4.3.2.2 DO Figures

4.3.2.2.1 DOWHILE. The flowchart for the DOWHILE control structure, which is a looping operation in which the test for loop termination precedes the code block, is as follows:



DOWHILE

All of the loops which are initiated by the PERFORM are of the structured programming DOWHILE format and consequently no simulation is needed. The statement is written as:

```
PERFORM PARAGRAPH-NAME UNTIL (P).
```

The conditional logic of the DOWHILE requires that the loop be terminated on a "false" condition so that in order to be compatible with structured programming definition, the test should read UNTIL (NOT p). However, since the flowchart of the PERFORM statement is that of a DOWHILE, the artificial negation of the termination logic in order to override the semantics of the word UNTIL is not necessary and if (p) were a complex condition, this negation could, in fact, be confusing.

The DOWHILE with indexing is also a part of the language and is written as:

```
PERFORM PARAGRAPH-NAME
    VARYING ID-1 FROM ID-2 BY ID-3
    UNTIL (P).
```

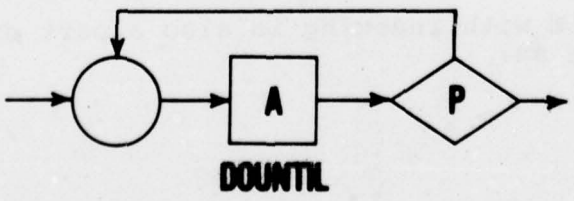
Note that while the indexing is varied automatically, the decision as to when to terminate the loop is still under control of the UNTIL rather than whether the indexing has reached some limit.

Finally, the option exists to PERFORM a block of code a specified number of times. This is written as:

```
PERFORM PARAGRAPH-NAME n TIMES.
```

Since, for all PERFORMS, the repetitive code must be placed out-of-line, there are no indentation requirements for these statements. However, for the indexed PERFORM (which can be written with up to three indexing variables), it is suggested that the format of the example shown in this subsection be adopted where words which control the indexing are indented further to the right than the UNTIL so that the conditions for loop termination are emphasized.

4.3.2.2.2 DOUNTIL. The DOUNTIL control structure is one in which the looping criteria are tested at the end of the loop and thus the code block is always executed at least once. The flowchart is as follows:



DOUNTIL

Since all PERFORMs are of the DOWHILE control logic, the DOUNTIL must be simulated. The recommended format is either:

```
PERFORM PARAGRAPH-NAME.
PERFORM PARAGRAPH-NAME UNTIL (p).
```

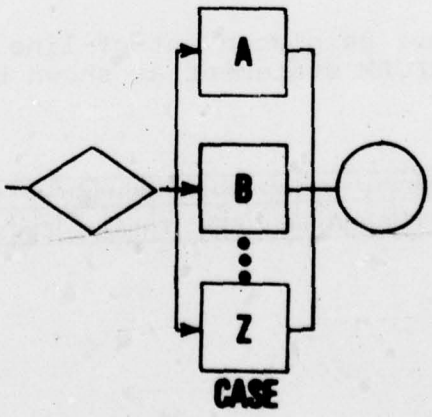
or

```
MOVE OFF-CODE TO LOOP ESCAPE;
PERFORM PARAGRAPH-NAME
    UNTIL LOOP-ESCAPE-ON.
    :
PARAGRAPH-NAME.
    IF (p)
        MOVE ON-CODE TO LOOP-ESCAPE.
        code a.
```

The latter case is most useful with a PERFORM that uses the VARYING option. However, the data item or index which is automatically varied by the execution of the PERFORM statement must never have its contents changed through the execution of a statement within the range of performed statements.

4.3.2.3 CASE Figure

The CASE figure causes control to be passed to one of a set of functional blocks of code (A,B,...,Z) based on the value of an integer variable. The flowchart for this figure is:



The CASE statement is not part of conventional COBOL and must therefore be simulated using the GO TO...DEPENDING ON statement. This verb permits the programmer to select one of a set of procedures depending upon the value of an integer whose range is from 1 to the number of procedure names listed in the statement. For any integer outside these limits the GO TO statement is ignored and control passes to the statement which follows it. This means that default code, if required, should immediately follow the GO TO...DEPENDING ON statement. At least one paragraph name is required. The simulation is as follows:

```
CASE-PARAGRAPH.
    GO TO CASE-1 CASE-2 ... DEPENDING ON I.
    default code.
    GO TO CASE-PARAGRAPH-END.
CASE-1.
    code A.
    GO TO CASE-PARAGRAPH-END.
CASE-2.
    code B.
    GO TO CASE-PARAGRAPH-END.

CASE-n.
    code Z.
CASE-PARAGRAPH-END.
    EXIT.
```

The above code must be placed out-of-line and is invoked by an in-line PERFORM statement as shown below:

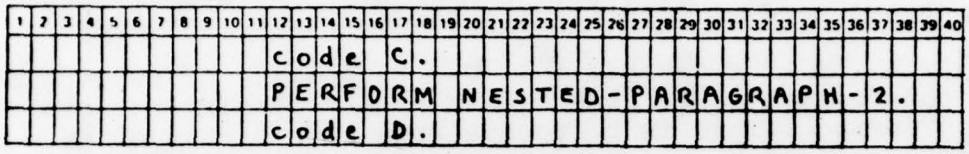
```
PERFORM CASE-PARAGRAPH THRU CASE-PARAGRAPH-END.
```

note that the PERFORM...THRU option of the PERFORM statement must be used because of the presence of the paragraph names which are used to identify the various case statements.

4.3.2.4 INCLUDE Capability. The capability of nesting blocks of code within other code blocks is a necessity for top down programming. This is best done when such blocks of code are stored and can be accessed by the COBOL COPY statement as separate members in a library system. However, it should be noted that this requirement is a compiler dependency and may not be possible for some ANS compilers. Since the COPY does not permit nesting, it is necessary to simulate this requirement with the use of nested PERFORM statements. The blocks of code which are PERFORMed are presumably stored as separate members which are easily accessed on a direct access device and are referenced for the COBOL compiler by means of COPY statements. This means that no COPY may appear in any block of code which is invoked by a copied PERFORM. With this technique the top level of the PROCEDURE DIVISION looks as follows:

```
PROCEDURE DIVISION.
TOP-PARAGRAPH.
    code A.
    PERFORM NESTED-PARAGRAPH-1.
    code B.
    STOP RUN.
NESTED-PARAGRAPH-1.
    COPY LIBRARY-NAME-1.
NESTED-PARAGRAPH-2.
    COPY LIBRARY-NAME-2.
```

Note that the COPY statement references library names, not paragraph names. "NESTED-PARAGRAPH-1" is a separate block of code which the COPY statement can access and may take the following form:



"NESTED-PARAGRAPH-2" is a sequence of statements similiar to those contained in the above paragraph within which it was invoked and it may contain other PERFORMs for deeper nesting. The COPY statements following the top paragraph insure that the compiler is aware of all the segments of code which comprise the total program. Furthermore, since no PERFORMed paragraph may contain a COPY, there is no danger of violating the nesting limitation of this verb.

4.3.3 Additional Recommended Coding Conventions

4.3.3.1 Restricted ANS COBOL Statement Usage. In order to preserve the concept of structured programming, it is recommended that the general usage of those statements in COBOL which permit changes of sequential control be restricted to an exception basis only, unless such statements are indicated in the standards in subsection 4.3.2 as a simulation requirement for the basic control logic figures.

4.3.3.2 Program Organization. The structure of a COBOL program is such that many of the rules for program organization have been predefined. For instance, all data must be specified in the DATA DIVISION. Furthermore, within this section, the formal rules which define permissible hierarchical data structures are sufficient to preserve the readability requirements of structured programming. However, within the PROCEDURE DIVISION, (with the exception of the DECLARATIVE SECTION) the rules of COBOL permit the ordering of the PERFORMed code blocks to be completely flexible.

For a development process in which no random access library exists, the ordering of the segments of PERFORMed COBOL paragraphs in the procedure division is more critical. This is because the source listing under this condition is a single sequential data set. At present, the suggested sequence is initially by nested level for two or three levels (depending on the program's complexity) and alphabetically thereafter.

PERFORMed paragraphs should be separated from the main body of code, and from other PERFORMed paragraphs, by at least two blank lines. Logically noncontiguous paragraphs (other than those used in the CASE figure) should be separated by at least one blank line.

4.3.3.3 Comments. One of the primary intents of the developers of the COBOL language was to produce a self-documenting language. When this is coupled with the discipline of structured programming the resulting programs should be even more readable. Experience has indicated that well written COBOL programs contribute toward meeting this objective. Therefore, it is recommended that the use of comments be held to a minimum. When they are used, they should be organized in such a manner as not to interfere with the readability of the program itself and should be grouped immediately before major subdivisions of code such as record description entries, Procedure Division paragraphs, etc. This may be done by such devices as using blank lines to insure that the comment stands apart from the program proper and starting and concentrating the textual commentary in the middle of the pages beginning in column 35 or 40.

4.3.3.4 Indentation and Formatting Conventions. Variables and structures defined in the DATA DIVISION should be arranged in a meaningful order. This order could be alphabetic, by class such as the days of the week, or any other class format. A suggested set of indentation rules for data items is as follows:

a. General Format. All level 77 and 01 variables should begin their level numbers in columns 10-11 and names starting in column 14. The PICTURE clause should be between columns 32-45, depending on the length of the longest variable name. All other clauses used should follow the PCITURE clause with normal spacing. If more than one line is needed for a variable's definition, the second and succeeding lines should be indented from the PICTURE clause as follows:

```
77  RECORD-COUNT                PICTURE 9(7)V99
                                COMP SYNC.
```

b. Structures. Each successively lower level in a structure should be indented four columns from the next higher level. Level numbers should be incremented by more than one and should precede each variable name in the structure on the same line and two columns before it as follows:

```
01  EMPLOYEE-RECORD.
    02  NAME.
        03  FIRST      PICTURE X(10).
        03  MIDDLE     PICTURE X.
        03  LAST       PICTURE X(20).
    02  ADDRESS.
        03  STREET     PICTURE X(15).
```

When conditioned-names (level 88 items) are used they should be indented and written with all value clauses vertically aligned.

```
    03  TYPE-CODE      PICTURE X.
        88  NEW-TYPE   VALUE 'B'.
        88  OLD-TYPE   VALUE '2'.
```

The indentation recommendations for the structured coding structures which appear in the PROCEDURE DIVISION were previously specified in the section where they were described. When programming the code blocks which represent the SEQUENCE in structured programming, each COBOL statement should start a new line. Any statement not subject to indentation rules starts in the same column in which the statement above it started. Only one statement per line is permitted. If any statement must be continued on the next line, the continued portion should be indented one column so as not to confuse it with structuring indentation.

```
    MOVE HOUR TO LAST-HOUR.
    ADD 1 TO HOUR.
    MULTIPLY MEASUREMENTS (HOUR) BY RATE-OF-DECAY
        GIVING RESIDUAL (HOUR).
    MOVE NEW-RATE TO RATE-OF-DECAY.
```

REFERENCES

1. Boehm, B. W., "Keynote Address: The High Cost of Software," TRW-SS-73-08, September 1973.

2. Boehm, B. W., et al, "Structured Programming: A Quantitative Assessment," IEEE Computer, June 1975, pages 38-54.

3. Boehm, B. W., "Structured Programming: Problems, Pitfalls, and Payoffs," TRW-SS-76-06, July 1976.

4. Dijkstra, E. W., "Notes on Structured Programming," in Structured Programming, O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoave, Academic Press, London, 1972.

5. IBM Corporation, An Introduction to Structured Programming in COBOL, Publication GCZO-1776-O, November 1975.

6. Kessler, Marvin and Tinanoff, N., "Structured Programming Series (Volume I), Programming Language Standards," RADC TR-74-300, Volume I, 15 March 1975, AD A016771.

7. Mills, H. D., "Mathematical Foundations for Structured Programming," IBM Federal Systems Division Technical Report FSC 72-6012, February 1972.

8. Smith, Paul, "Fortran Code Auditor - User's Manual," RADC-TR-76-395, Volume I, December 1976.

DISTRIBUTION

Addressees	Copies
CCTC Codes	
C100	1
C110	40
C124 (Reference & Record Sets)	3
C124 (Stock)	6
C126 (Library)	2
C200	2
C300	10
C400	10
C600	2
C700	8
DCA Codes	
101R	1
205	2
630	2
Other	
DCAOC/N300	2
DCEC/R800	2
Defense Documentation Center, Cameron Station, Alexandria, Virginia 22314	12
	135

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TM 185-78	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Guidelines for Structured Coding		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Captain John R. Scott		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Command and Control Technical Center The Pentagon Washington, D.C. 20301		12. REPORT DATE 1 September 1978 13. NUMBER OF PAGES 51 pages
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.
Copies of this document may be obtained from the
Defense Documentation Center, Cameron Station,
Alexandria, Virginia 22314

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This manual provides guidelines for DCA/CCTC personnel writing computer programs using structured code. The guidelines are for handcoding of structured constructs in FORTRAN and COBOL, with additional instructions on module sizing and formatting.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE 7-1

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)